Genetic Programming and Evolvable Machines (2021) 22:511–537
https://doi.org/10.1007/s10710-021-09412-w

EvoStencils: a grammar-based genetic programming approach for constructing efficient geometric multigrid methods

Jonas Schmitt[1] · Sebastian Kuckuk[1] · Harald Köstler[1]

Received: 30 April 2021 / Revised: 17 August 2021 / Accepted: 21 August 2021 /
Published online: 3 September 2021

Abstract
For many systems of linear equations that arise from the discretization of partial differential equations, the construction of an efficient multigrid solver is challenging. Here we present EvoStencils, a novel approach for optimizing geometric multigrid methods with grammar-guided genetic programming, a stochastic program optimization technique inspired by the principle of natural evolution. A multigrid solver is represented as a tree of mathematical expressions that we generate based on a formal grammar. The quality of each solver is evaluated in terms of convergence and compute performance by automatically generating an optimized implementation using code generation that is then executed on the target platform to measure all relevant performance metrics. Based on this, a multi-objective optimization is performed using a non-dominated sorting-based selection. To evaluate a large number of solvers in parallel, they are distributed to multiple compute nodes. We demonstrate the effectiveness of our implementation by constructing geometric multigrid solvers that are able to outperform hand-crafted methods for Poisson's equation and a linear elastic boundary value problem with up to 16 million unknowns on multi-core processors with Ivy Bridge and Broadwell microarchitecture.

Keywords Grammar-guided genetic programming · Multigrid methods · Algorithm design · Codegeneration · Distributed computing

Jonas Schmitt
jonas.schmitt@fau.de

Sebastian Kuckuk
sebastian.kuckuk@fau.de

Harald Köstler
harald.koestler@fau.de

[1] Chair for System Simulation, Friedrich-Alexander University Erlangen-Nürnberg, Erlangen, Germany

1 Introduction

Solving the linear or nonlinear systems that arise from the discretization of partial differential equations efficiently is an unprecedented challenge. The vast number of unknowns in many of these systems necessitates the design of fast and scalable solvers. Unfortunately, the optimal solution method highly depends on the system itself, and it is infeasible to formulate a single algorithm that works efficiently in all cases. Geometric multigrid methods are a class of asymptotically optimal multilevel solution algorithms for (non-)linear systems, which were first formulated by Fedorenko in 1961 [12] and have been later pioneered by Brandt [4] and Hackbusch [16]. These methods are based on accelerating the convergence of stationary iterative methods by applying corrections obtained on a lower resolution of the original problem. A comprehensive overview of multigrid methods can be found in [5, 34]. Even though, since the invention of multigrid, significant effort has been put into the design of efficient solvers for many important cases, such as Helmholtz [11] and saddle point problems [2], this task is still an open challenge.

Oosterlee et al. [27] already considered the optimization of multigrid solvers by choosing each component from a finite number of options. The resulting discrete optimization problem is then solved using a genetic algorithm. Similarly, the work by Thekale et al. [33] aims to optimize the number of full multigrid (FMG) cycles using a branch-and-bound approach while the recent work by Brown et al. treats the optimization of a solver's parameter as a minimax problem [6]. These approaches have in common that they impose certain constraints on a solver's structure and then aim to find the optimal set of options under these conditions. Thekale et al. consider an FMG solver consisting of V-cycles and then focuses on finding the optimal number of cycles. While Osterlee et al. and Brown et al. consider a larger number of algorithmic parameters, the optimization is still restricted to cycles of a particular structure and, therefore, lacks the flexibility to adapt the individual steps of the algorithm independent from each other. For instance, they do not consider the combination of different smoothers, prolongation, restriction, and cycles on each level.

To overcome these limitations, we treat the task of finding an optimal multigrid solver as an algorithm design problem by proposing a novel context-free grammar for the automatic generation of arbitrary sequences of multigrid operations, which we have first formulated in [32]. This approach allows us to alter each step performed within the algorithm by expressing it in a separate production rule. Based on the order and choice of productions, we can construct arbitrarily composed multigrid cycles that combine the different building blocks of these methods to push the boundaries of classical parameter optimization methods, such as [6, 27, 33]. In [32], we could show how convergence and performance estimates can be automatically obtained for geometric multigrid solvers on rectangular grids of arbitrary size and how, based on these metrics, a multi-objective optimization can be performed using genetic programming (GP) [23] and a covariance matrix adaptation evolution strategy (CMA-ES) [17]. However, the resulting estimates could not yield sufficient accuracy in all investigated cases and thus have, so far, limited the outcome of the optimization. Here, we demonstrate an extension of this approach that replaces the prediction obtained from mathematical analysis and performance modeling by a code generation-based evaluation that can be carried out sufficiently fast by distributing its computation to multiple compute nodes.

A different direction within the optimization of multigrid methods, which has recently become popular, is applying machine learning to improve the individual solver components, such as [15, 19, 20, 22]. While Greenfeld et al. [15], Katrutsa et al. [22], and Huang et. al [20] focus on learning efficient prolongation operators or smoothers, the work by Hsieh et al. [19] aims to improve an existing solver through the supplemental application of a neural network. Consequently, these works regard the composition of a solver as immutable but instead focus on optimizing its components or improving the outcome of an existing method through additional update steps. In contrast, our approach considers each building block of a solver as a black box and aims to find an optimal composition.

The paper is structured as follows. In the first step, we present the derivation of our context-free grammar for the automatic construction of geometric multigrid solvers, which forms the basis for all subsequent sections. Next, we present the extension of our previous work by a distributed code generation-based evaluation and describe the resulting grammar-guided evolutionary search method. Finally, we demonstrate our approach's effectiveness by constructing efficient multigrid solvers for a similar linear elastic boundary value problem as considered in [32] and two Poisson problems.

2 A formal grammar for constructing multigrid solvers

The task of constructing a multigrid solver for a particular problem is typically performed by a human expert with profound knowledge in numerical mathematics. To automate this task, we first need to represent multigrid solvers in a formal language that we can then use to construct different instances on a computer. The rules of this language must ensure that only valid solver instances can be defined, which means that we can automatically determine their convergence speed and execution time. Additionally, we want to enforce that the generated method works on a hierarchy of grids, which requires the availability of inter-grid operations to obtain approximations of the same operator or grid on a finer or coarser level. Consider the general system of linear equations defined on a grid with spacing h

$$A_h u_h = f_h, \tag{1}$$

where A_h is the coefficient matrix, u_h the unknown and f_h the right-hand side of the system. Each component of a multigrid solver can be written in the form

$$u_h^{i+1} = u_h^i + \omega B_h (f_h - A_h u_h^i), \tag{2}$$

where u_h^i is the approximate solution in iteration i, $\omega \in \mathbb{R}$ the relaxation factor and B_h an operator defined on the given level with spacing h. For example, with the splitting $A_h = D_h - U_h - L_h$, where D_h represents the diagonal, $-U_h$ the upper triangular and $-L_h$ the lower triangular part of A_h, we can define the Jacobi

$$u_h^{i+1} = u_h^i + D_h^{-1}(f_h - A_h u_h^i) \tag{3}$$

and the lexicographical Gauss-Seidel method

$$u_h^{i+1} = u_h^i + (D_h - L_h)^{-1}(f_h - A_h u_h^i). \tag{4}$$

If we assume the availability of a restriction operator I_h^H, that computes an approximation of the residual on a coarser grid with spacing H, a prolongation operator I_H^h, that interpolates a correction obtained on the coarser grid into a finer grid, and an approximation for the inverse of A_h on the coarser grid, a coarse grid correction can be defined as

$$u_h^{i+1} = u_h^i + I_H^h A_H^{-1} I_h^H (f_h - A_h u_h^i). \tag{5}$$

Furthermore, we can substitute u_h^i in (5) with (3) and obtain a two grid with Jacobi pre-smoothing

$$\begin{aligned} u_h^{i+1} &= (u_h^i + D_h^{-1}(f_h - A_h u_h^i)) \\ &\quad + I_H^h A_H^{-1} I_h^H (f_h - A_h(u_h^i + D_h^{-1}(f_h - A_h u_h^i))). \end{aligned} \tag{6}$$

By repeatedly substituting subexpressions, we can automatically construct a single expression for any multigrid solver. If we take the set of possible substitutions as a basis, we can define a list of rules according to which such an expression can be generated. We specify these rules in the form of a context-free grammar, which is described in Table 1. Table 1a contains the production rules while Table 1b describes their semantics. Within the former the symbol A_h^+ corresponds to a given splitting of the system matrix $A_h = A_h^+ + A_h^-$ such that A_h^+ is efficiently invertible. For instance, in case of the Jacobi method $A_h^+ = D_h$ is defined as the diagonal of A_h. Each rule defines the set of expressions by which a certain production symbol, denoted by $\langle \cdot \rangle$, can be replaced. Starting with $\langle S \rangle$, symbols are recursively replaced until the produced expression contains only terminals or the empty string λ. The construction of a multigrid solver comprises the recursive generation of cycles on multiple levels. Consequently, it must be possible to create a new system of linear equations on a coarser level, including a new initial solution, right-hand side, and coefficient matrix. Moreover, if we decide to finish the computation on a particular level, we need to restore the state of the next finer level, i.e., the current solution and right-hand side, when applying the coarse grid correction. The current state of a multigrid solver on a level with grid spacing h is represented as a tuple (u_h, f_h, δ_h), where u_h represents the current iterate, f_h the right-hand side and δ_h a correction expression. To restore the current state on the next finer level, we additionally include a reference $state_h$ to the corresponding tuple. According to Table 1a, the construction of a multigrid solver always ends when the tuple $(u_h^0, f_h, \lambda, \lambda)$ is reached. This tuple contains the initial solution and right-hand side on the finest level and therefore corresponds to the original system of linear equations that we aim to solve.

Table 1 Formal grammar for constructing three-grid multigrid cycles—The first column contains the list of production rules where each symbol on the left side of the ⊨ sign can be replaced by the corresponding symbol on its right side

$\langle S \rangle$	$\models$	$\langle s_h \rangle$
$\langle s_h \rangle$	$\models$	$\textsc{iterate}(\langle c_h \rangle,\ \omega,\ \langle \mathcal{P} \rangle)$
$\langle s_h \rangle$	$\models$	$\textsc{iterate}(\textsc{apply}(\langle B_h \rangle,\ \langle c_h \rangle),\ \omega,\ \langle \mathcal{P} \rangle)$
$\langle s_h \rangle$	$\models$	$\textsc{iterate}(\textsc{cgc}(I_{2h}^{h},\ \langle s_{2h} \rangle),\ \omega,\ \langle \mathcal{P} \rangle)$
$\langle s_h \rangle$	$\models$	$(u_h^0,\ f_h,\ \lambda,\ \lambda)$
$\langle c_h \rangle$	$\models$	$\textsc{residual}(A_h,\ \langle s_h \rangle)$
$\langle B_h \rangle$	$\models$	$\textsc{inverse}(A_h^+)$ with $A_h = A_h^+ + A_h^-$
$\langle c_{2h} \rangle$	$\models$	$\textsc{residual}(A_{2h},\ \langle s_{2h} \rangle)$
$\langle c_{2h} \rangle$	$\models$	$\textsc{cocy}(A_{2h},\ u_{2h}^0,\ \textsc{apply}(I_h^{2h},\ \langle c_h \rangle))$
$\langle s_{2h} \rangle$	$\models$	$\textsc{iterate}(\langle c_{2h} \rangle,\ \omega,\ \langle \mathcal{P} \rangle)$
$\langle s_{2h} \rangle$	$\models$	$\textsc{iterate}(\textsc{apply}(\langle B_{2h} \rangle,\ \langle c_{2h} \rangle),\ \omega,\ \langle \mathcal{P} \rangle)$
$\langle s_{2h} \rangle$	$\models$	$\textsc{iterate}(\textsc{apply}(I_{4h}^{2h},\ \langle c_{4h} \rangle),\ \omega,\ \lambda)$
$\langle B_{2h} \rangle$	$\models$	$\textsc{inverse}(A_{2h}^+)$ with $A_{2h} = A_{2h}^+ + A_{2h}^-$
$\langle c_{4h} \rangle$	$\models$	$\textsc{apply}(A_{4h}^{-1},\ \textsc{apply}(I_{2h}^{4h},\ \langle c_{2h} \rangle))$
$\langle \mathcal{P} \rangle$	$\models$	$\textsc{partitioning} \mid \lambda$

(a) Production rules

```
function ITERATE($(u,\ f,\ \delta,\ state),\ \omega,\ \mathcal{P}$)
    $\tilde{u} \leftarrow u + \omega \cdot \delta$ with $\mathcal{P}$
    return $(\tilde{u},\ f,\ \lambda,\ state)$
end function
function APPLY($B,\ (u,\ f,\ \delta,\ state)$)
    $\tilde{\delta} \leftarrow B \cdot \delta$
    return $(u,\ f,\ \tilde{\delta},\ state)$
end function
function RESIDUAL($A,\ (u,\ f,\ \lambda,\ state)$)
    $\delta \leftarrow f - Au$
    return $(u,\ f,\ \delta,\ state)$
end function
function COCY($A_H,\ u_H^0,\ (u_h,\ f_h,\ \delta_H,\ state_h)$)
    $u_H \leftarrow u_H^0$
    $f_H \leftarrow \delta_H$
    $\tilde{\delta}_H \leftarrow f_H - A_H u_H^0$
    $state_H \leftarrow (u_h,\ f_h,\ \lambda,\ state_h)$
    return $(u_H,\ f_H,\ \tilde{\delta}_H,\ state_H)$
end function
function CGC($I_H^h,\ (u_H,\ f_H,\ \lambda,\ state_H)$)
    $(u_h,\ f_h,\ \lambda,\ state_h) \leftarrow state_H$
    $\tilde{\delta}_h \leftarrow I_H^h \cdot u_H$
    return $(u_h,\ f_h,\ \tilde{\delta}_h,\ state_h)$
end function
```

(b) Semantics

The occurrence of the same symbol at the left side of different rules means that multiple alternative productions can be applied to this symbol. Finally, the list of functions in the second column formally specifies the semantic behavior of each symbol

Here we have neither computed a correction nor need to restore the state, and both δ_h and $state_h$ contain the empty string.

In general, our grammar includes three functions that operate on a fixed level. The function ITERATE generates a new state tuple based on the previous one by applying the correction δ to the current iterate u using the relaxation factor ω. If available, a partitioning can be included to perform the update in multiple sweeps on subsets of u and δ, for example, a red-black Gauss-Seidel iteration. The function RESIDUAL creates a residual expression based on the given state, which is assigned to the newly created symbol δ. A correction δ can be transformed with the function APPLY, which generates a new correction $\tilde{\delta}$ by applying the linear operator B to the old one. For example, the following function applications evaluate to one iteration of damped Jacobi smoothing:

$$
\begin{aligned}
&\text{ITERATE}(\text{APPLY}(D_h^{-1},\ \text{RESIDUAL}(A_h,\ (u_h^0,\ f_h,\ \lambda,\ \lambda))),\ 0.7,\ \lambda)\\
\rightarrow\ &\text{ITERATE}(\text{APPLY}(D_h^{-1},\ (u_h^0,\ f_h,\ f_h - A_h u_h^0,\ \lambda)),\ 0.7,\ \lambda)\\
\rightarrow\ &\text{ITERATE}((u_h^0,\ f_h,\ D_h^{-1}(f_h - A_h u_h^0),\ \lambda),\ 0.7,\ \lambda)\\
\rightarrow\ &(u_h^0 + 0.7 \cdot D_h^{-1}(f_h - A_h u_h^0),\ f_h,\ \lambda,\ \lambda)
\end{aligned}
$$

Finally, it remains to be shown how one can recursively create a multigrid cycle on the next coarser level and then apply the result of its computation to the current approximate solution. This is accomplished through the functions COCY and CGC[1]. The former expects a state to which the restriction I_h^H has been already applied. It then creates a new state on the next coarser level using the initial solution u_H^0, the operator A_H, and the restricted correction δ_H as a right-hand side f_H. Note that on the coarsest level, the resulting system of linear equations can be solved directly, which is denoted by the application of the inverse coarse-grid operator. For restoring the previous state, a reference is stored in $state_H$. If the computation on the coarser level is finished, the function CGC comes into play. It first restores the previous state on the next finer level and then computes a coarse-grid correction by applying the prolongation operator to the solution computed on the coarser grid, which is then used as a new correction $\tilde{\delta}_h$ on the finer level. Again the following example application demonstrates the semantics of these functions:

$$\begin{aligned}
&\text{CGC}(I_{2h}^h, \text{ITERATE}(\text{COCY}(A_{2h},\ u_{2h}^0,\ (u_h^0,\ f_h,\ I_h^{2h}(f_h - A_h u_h^0),\ \lambda)),\ 1,\ \lambda)) \\
\rightarrow\ &\text{CGC}(I_{2h}^h,\ \text{ITERATE}((u_{2h}^0,\ I_h^{2h}(f_h - A_h u_h^0),\ I_h^{2h}(f_h - A_h u_h^0) - A_{2h} u_{2h}^0, \\
&(u_h^0,\ f_h,\ \lambda,\ \lambda)),\ 1,\ \lambda)) \\
\rightarrow\ &\text{CGC}(I_{2h}^h,\ (u_{2h}^0 + 1 \cdot (I_h^{2h}(f_h - A_h u_h^0) - A_{2h} u_{2h}^0),\ I_h^{2h}(f_h - A_h u_h^0),\ \lambda, \\
&(u_h^0,\ f_h,\ \lambda,\ \lambda))) \\
\rightarrow\ &(u_h^0,\ f_h,\ I_{2h}^h \cdot (u_{2h}^0 + 1 \cdot (I_h^{2h}(f_h - A_h u_h^0) - A_{2h} u_{2h}^0)),\ \lambda)
\end{aligned}$$

Finally, note that Table 1a can produce multigrid cycles with a hierarchy of at most three discretization levels (or coarsening steps), whereas the only viable operation on the lowest level is the application of a coarse grid solver. However, since its rules can be applied recursively, the depth of the resulting grammar expression tree is not restricted, and, in principle, all three discretization levels can be traversed an infinite number of times. In practice, it is often favorable to construct multigrid solvers that employ an even greater number of coarsening steps. For this purpose, additional production rules for the generation of inter-grid transfer operations, i.e., COCY and CGC, must be defined on the respective discretization levels, whereas the general structure of the grammar remains unchanged. Since we have shown how it is possible to generate expressions that uniquely represent different multigrid solvers using the formal grammar defined in Table 1, this paper's remainder focuses on the evaluation and optimization of the algorithms resulting from this representation.

3 Optimization objectives and search space estimation

The efficiency of an iterative method for solving a given problem is defined by two objectives: its convergence rate and compute performance on the target platform. This work focuses on the automatic optimization of geometric multigrid solvers on

[1] The names of these functions represent abbreviations for the terms coarse cycle and coarse-grid correction, respectively.

rectangular grids. In this case, we can represent all matrices as one or multiple stencils, which facilitates the application of both predictive models for convergence and performance predictions as well as the utilization of techniques for the automatic generation and domain-specific optimization of scalable solver implementation. In the following, we first give an overview of the possibilities and limitations of predicting a solver's convergence and compute performance based on mathematical analysis and performance modeling. We then explain how the inherent limitations of these techniques can be overcome with a distributed code generation-based solver evaluation and optimization using grammar-guided genetic programming. Finally, we conclude the description of our optimization approach with implementation details about EvoStencils,[2] an open source Python tool for the grammar-guided optimization of geometric multigrid methods.

3.1 Convergence estimation

The quality of an iterative method is first and foremost determined by its convergence rate, i.e., the speed at which the approximation error approaches machine precision. One iteration of a multigrid solver can be expressed in the general form of Eq. (2). By separating all terms that contain the current iterate u_h^i from the rest of the equation, we obtain the following form:

$$u_h^{i+1} = (I_h - \omega B_h A_h) u_h^i + \omega B_h f_h, \tag{7}$$

where I_h is the unit matrix. The *iteration matrix* M_h of the multigrid solver is then given by

$$M_h = (I_h - \omega B_h A_h). \tag{8}$$

The *spectral radius* ρ of this matrix, as defined by

$$\rho(M_h) = \max_{1 \leq j \leq n} |\lambda_j(M_h)|, \tag{9}$$

where $\lambda_j(M_h)$ are the eigenvalues of M_h, is essential for the convergence of the method. Assume u_h^* is the exact solution of the system, the error $e_h^i = u_h^i - u_h^*$ in iteration i then satisfies,

$$e_h^i = (M_h)^i e_h^0, \tag{10}$$

where $(M_h)^i$ is the ith power of M_h. The *convergence factor* of this sequence is the limit

[2] EvoStencils: https://github.com/jonas-schmitt/evostencils.

$$\rho = \lim_{i \to \infty} \left(\frac{\left\| e_h^i \right\|}{\left\| e_h^0 \right\|} \right)^{1/i}, \tag{11}$$

which is equal to the spectral radius of the iteration matrix M_h [29]. In general, the computation of the spectral radius is of complexity $\mathcal{O}(n^3)$ for $M_h \in \mathbb{R}^{n \times n}$ [9]. If we, however, restrict ourselves to geometric multigrid solvers on rectangular grids, we can employ local Fourier analysis (LFA) to obtain an estimate for ρ [35]. LFA considers the original problem on an infinite grid while the boundary conditions are neglected. Recently LFA has been automated through the use of software packages [21, 28]. To automatically estimate a multigrid solver's convergence factor, we first need to obtain the iteration matrix. Using the grammar described in the last section, we consistently generate expressions of the form of Eq. (2) from which we can extract the iteration matrix by transforming it to the representation formulated in Eq. (7). Finally, we can emit the resulting combined expression, representing the iteration matrix of a complete multigrid solver for which the spectral radius can be estimated using automated local Fourier analysis.

3.2 Compute performance estimation

A popular yet simple model for estimating an algorithm's performance on modern computer architectures is the *roofline model* [36]. Based on the operational intensity of a compute kernel, i.e., the ratio of floating-point operations to words loaded from and stored to memory, it estimates the maximum achievable performance, which is either limited by the memory bandwidth or the compute capabilities of the machine. The basic roofline formula is given by

$$P = \min(P_{max}, \ I \cdot b_s), \tag{12}$$

where P is the attainable performance, P_{max} the peak performance of the machine, i.e., the maximum achievable amount of floating-point operations per second, I the operational intensity of the kernel, and b_s the peak bandwidth, i.e., the number of words that can be moved from and to main memory in every second. Each operation within a geometric multigrid solver either represents a matrix-vector or vector-vector operation, where each vector corresponds to a regular grid and each matrix to one or multiple stencil codes. If we explicitly represent each operation in the form of a stencil, the computation of the operational intensity is straightforward.

3.3 Search space estimation

To find the optimal geometric multigrid solver for a specific problem, the structure and size of the search space dictate what types of optimization methods can be applied. With a sufficiently small search space, one could attempt to enumerate all possible solutions. This approach's infeasibility becomes apparent when looking at the grammar in Table 1. Assume our goal is to find a multigrid solver that operates on three levels, but the only allowed operation on the coarsest level is applying a direct solver. Now assume we want to evaluate all solvers that perform at least one but at most six smoothing steps on each level, whereby in each step, we can choose a different smoother from four alternatives, each of them available with or without a red-black partitioning of the computation. Moreover, we require that a coarse-grid correction is always performed before smoothing and that the number of coarse-grid corrections never exceeds the number of smoothing steps. Consequently, for each step of our multigrid solver, we must choose from $4 = 2^2$ different smoothers while we further need to decide if we want to apply a red-black partitioning and if we want to perform a coarse-grid correction beforehand, which results in a total number of 2^4 choices. Since we also want to consider all possible configurations that perform more than one but less than six smoothing steps on each level, the size of the search space is approximately $\sum_{k=2}^{12} 2^{4k} \approx 3 \cdot 10^{14}$. Consequently, we have to consider $3 \cdot 10^{14}$ alternatives that all need to be evaluated for both objectives. If we assume that evaluating a particular solver for both objectives takes on average ten milliseconds on a multi-core CPU, even a supercomputer consisting of one million such processors would require more than 30 days to evaluate all possible alternatives. This number will be even higher in practice, especially if we consider more levels and configuration options, which renders a brute-force approach practically impossible.

Note that in [32] we have treated the choice of relaxation factors ω as an additional continuous optimization problem, while within the construction of multigrid solver expressions, each relaxation factor has been first fixed to a default value of 1. However, if it is possible to predict the method's convergence factor accurately, it is beneficial to target both optimization problems at once. The reason for this is that certain combinations of smoothers and coarse-grid corrections only lead to a converging solver in combination with over- or underrelaxation, i.e., the choice of a relaxation factor smaller or larger than one, respectively. For instance, the Jacobi method often only represents an efficient smoother when underrelaxation is used [34]. Consequently, considering the choice of relaxation factors as a separate optimization problem comprises the risk of a premature eviction of solver components that require over- or underrelaxation for their functioning. In contrast to [32], we, therefore, choose each relaxation factor that occurs within one of the productions in Table 1a randomly from an evenly-spaced interval, which increases the size of the search space even further.

4 Grammar-guided evolutionary search method

If the search space is too large to be directly enumerated, a remedy is to use heuristics to search efficiently through the space of possible solutions and still find the global or at least a local optimum. Evolutionary algorithms are a class of search heuristics inspired by the principle of natural evolution that have been successfully applied to numerous domains [24]. These methods have in common that they evolve a population of solutions (called individuals) through the iterative application of so-called genetic operators. The order and probability of application of each operation can be varied, and different choices have been suggested for different optimization problems [1]. The exact implementation of each genetic operator depends on the problem class, i.e., the solution's structure. Our goal is to find the list of productions that, according to the context-free grammar shown in Table 1, leads to the optimal multigrid solver. The class of evolutionary algorithms that optimize expressions according to formal grammars is summarized under the term grammar-guided (or grammar-based) genetic programming (GGGP) [26]. In principle, our goal is to construct a solver with minimal execution time for reducing the approximation error of a given problem down to the required tolerance. While in [32] we could only predict this metric, a distributed evaluation of the considered solvers enables us to measure it directly.

4.1 Code generation and parallel evaluation

Even though the application of predictive models to estimate the convergence speed and compute performance of a grammatically represented solver has several advantages, the experiments performed in [32] indicate that, in practice, this approach does not consistently achieve sufficient accuracy for identifying solvers that outperform hand-crafted methods. An alternative approach is to employ code generation to automatically generate the optimized implementation of a solver, which can be executed on a modern computer architecture to extract all relevant performance metrics. For this purpose, we employ the ExaStencils[3] code generation framework, which was specifically designed for the generation of geometric multigrid implementations that run on parallel and distributed systems [25]. First, we transform the evolved multigrid expression to an algorithmic representation, which is then emitted in the form of a specification in ExaStencils' domain-specific language (DSL). Based on this representation the framework generates a C++ implementation of the solver which we finally run on a representative problem instance to measure both its total execution time and defect reduction factor

$$\tilde{\rho}_i = \frac{\left\| f_h - A_h u_h^i \right\|}{\left\| f_h - A_h u_h^{i-1} \right\|} \tag{13}$$

per iteration i on the target platform. We then obtain an approximate for the asymptotic convergence factor

$$\tilde{\rho} = \left(\prod_{i=1}^{n} \tilde{\rho}_i \right)^{1/n}, \tag{14}$$

where n is the number of iterations until convergence [34]. To cope with computational expense of performing this process for each solver considered within an optimization, we distribute its execution to multiple compute nodes such that it can be performed in parallel. With the availability of sufficient computational resources, we can, thus, perform an optimization that is purely based on a direct evaluation of all

[3] ExaStencils: https://www.exastencils.fau.de.

considered solvers and, hence, does not rely on the accuracy of a model-based prediction. Furthermore, while the complexity of an LFA-based prediction of a solver's convergence grows exponentially with the number of coarse-grid correction steps, the time required for code generation only increases linearly with the number of subexpressions within the grammatical representation of a multigrid solver. It is, hence, possible to evaluate multigrid solvers that operate on a grid hierarchy with significantly larger depth, for instance, a five-grid method.

4.2 Identification of optimal solvers

The execution time of a solver depends on its performance on the computer architecture employed for this measurement. Consequently, even though modern parallel architectures share certain commonalities, a solver with minimal execution time on specific evaluation hardware does not necessarily achieve the same performance on a different platform. Since our goal is to find solvers that are efficient for a wide range of modern architectures, a single-objective optimization that minimizes the time required for solving the given problem on the evaluation platform is insufficient. However, suppose we assume that a specific solver achieves faster convergence and a lower execution time per iteration than another one on particular hardware. In that case, there is a high probability that it will also achieve the same result on similar computer architectures. As in [32] we, therefore, treat the construction of an optimal multigrid solver as a multi-objective optimization problem, whereas we replace the model-based predictions used therein by the measured values of the convergence factor and execution time per iteration for a solver on the evaluation platform. To evolve a Pareto front of multigrid solvers, we employ a non-dominated sorting-based selection [7]. We expect that all solvers that turn out to be Pareto-optimal for these two objectives on the evaluation platform will also represent efficient solvers on similar computer architectures. Consequently, to identify an optimal solver, it is sufficient to consider only those contained in the Pareto front obtained with optimization on the evaluation hardware. If the amount of Pareto-optimal solvers is large, we can then further restrict the number of considered solvers, for instance, by sorting them according to their required solving time on the evaluation hardware.

4.3 Implementation details

To accomplish all steps from the automatic construction of arbitrarily composed multigrid solver expressions to the generation of scalable implementations on various computer architectures and the identification of Pareto-optimal solvers, a flexible implementation is needed that combines different programming languages and libraries under a common framework. For this purpose, we have created EvoStencils, an open source Python tool for the grammar-guided optimization of multigrid methods, whose software architecture is shown in Fig. 1. First, our implementation extracts all required information about the system matrix, solution field, and associated right-hand side from a formulation in ExaStencil's DSL ExaSlang [30, 31]. Based on this information, a context-free grammar similar to Table 1 is automatically

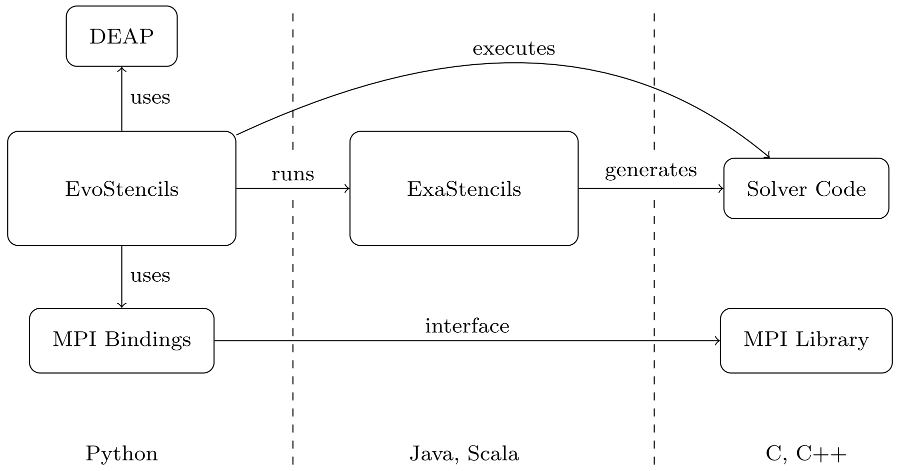

Fig. 1 Software Architecture of EvoStencils

constructed based on the GP module of the framework DEAP [13]. Each expression tree generated through GGGP is transformed to the graph-based internal representation of a multigrid solver and then evaluated using code generation, as described in Sect. 4.1. Since this functionality is fully encapsulated in Python functions, we can use all available optimization algorithms already implemented within DEAP. While EvoStencils is implemented in pure Python, the MPI library is accessed through language bindings to distribute the process of solver generation and evaluation to multiple compute nodes. Since MPI is an established standard for parallel computing on multi-node systems, this allows us to run EvoStencils on most of today's clusters and supercomputers.

5 Experiments

In [32] we could already demonstrate the construction of functioning multigrid solvers for a linear elastic boundary value problem based on a prediction-guided optimization. While we were able to show that our optimization approach is more efficient than a random search, the multigrid solvers obtained with it were not able to outperform hand-crafted methods for the given test case. In this work, building upon this, we aim to overcome these limitations through a distributed code generation-based solver evaluation. For consistency, we consider the same linear elastic boundary value problem as in [32], but also evaluate our approach on two- and three-dimensional Poisson problems. Poisson's equation is a well-studied PDE in multigrid theory and practice, facilitating the interpretability of the results obtained on these problems. Each of the resulting linear systems is considered solved when the initial defect is reduced by a factor of 10^{-12}. In the following, we first present the considered multigrid solver components and the general configuration of our optimization algorithm used within all subsequent experiments.

Table 2 Summary of GGGP configuration parameters

Parameter	Value
Evolutionary algorithm type	$(\mu + \lambda)$
Genetic programming variant	Tree-based
Number of objectives	2
Number of generations	250
Initial population size	2048
λ	256
μ	256
Number of MPI processes	64
Non-dominated sorting procedure	[14]
Selection operator	[8]
Crossover operator	Single-point crossover
Crossover probability	2/3
Mutation operator	Random subtree replacement
Probability to mutate a terminal symbol	1/3

5.1 Optimization settings

Within all optimization runs we choose a step size of $h = 1/2^l$ on each level l, whereby we employ a range of $l \in \left[l_{max} - 4, l_{max}\right]$. As such our goal is to construct an optimal five-grid method for the given problem. We then consider the following components for the construction of a multigrid solver:

Smoothers:	Decoupled/Collective Jacobi and red-black Gauss-Seidel, block Jacobi with rectangular blocks up to a maximum number of 6 terms.
Restriction:	Full-weighting restriction.
Prolongation:	Bilinear interpolation.
Relaxation factors:	$\omega \in (0.1 + 0.05i)_{i=0}^{36} = (0.1, 0.15, 0.2, \dots 1.9)$
Coarse grid solver:	Conjugate gradient method for $l = l_{max} - 4$.

To generate block Jacobi smoothers, we define a splitting $A = L + D + U$ where D is a block diagonal matrix, such that we have to solve a local system whose size corresponds to the size of a block at every grid point. For a more detailed treatment of block relaxation methods, the reader is referred to [34]. The relaxation factor ω for each smoothing and coarse grid correction step is chosen from the above sequence.

Table 2 contains a summary of the parameters used in our implementation of GGGP. To obtain a Pareto front of multigrid expressions, starting with a randomly initialized population of 2048 individuals, we perform a multi-objective optimization for 250 generations using tree-based GGGP implemented as a $(\mu + \lambda)$ evolution strategy [3], where we choose $\mu = \lambda = 256$ and employ the non-dominated sorting procedure presented in [14] (NSGA-II). Hence in each generation, we create λ individuals based on an existing population of size μ and then select the best

Table 3 Considered Poisson problem instances

Problem	2D Poisson	3D Poisson
Ω	$(0,1)^2$	$(0,1)^3$
$f(\mathbf{x})$	$\pi^2\cos(\pi x) - 4\pi^2\sin(2\pi y)$	$x^2 - 0.5y^2 - 0.5z^2$
$g(\mathbf{x})$	$\cos(\pi x) - \sin(\pi y)$	0

μ individuals for the next generation from the combined set. Each individual's fitness consists of two objectives: the asymptotic convergence factor $\tilde{\rho}$ and its execution time per iteration t both measured using a code generation-based evaluation, as described in 4.1. Here, we make use of the following optimization flags of the ExaStencils code generator: *opt_useAddressPrecalc*, *opt_loopCarriedCSE* and *opt_vectorize*, which enable an automatic address precalculation, common subexpression elimination and vectorization, respectively. We also enable the inversion of local matrices that occur within certain smoothers during within code generation by setting the flag *experimental_resolveLocalMatSys* accordingly. To compile the resulting C++ solver, we employ the GCC compiler with version 9.3.0 using one OpenMP thread per physical CPU core. We execute each solver three times and compute the resulting average for both objectives to reduce the influence of temperature and manufacturing-based variations in CPU performance.

Individuals are selected for crossover and mutation using a dominance-based tournament selection as described in [8]. New individuals are created by either crossover with a probability of 2/3, whereby we employ single-point crossover, or by mutation, either through replacement of a certain subtree with a new randomly created one, with a relative probability of 2/3, or through replacement of a single terminal or non-terminal symbol with a randomly chosen alternative. To evaluate $\lambda = 256$ individuals in each generation, we employ 64 MPI processes executed on 32 nodes of the Meggie Cluster of the Erlangen National High Performance Computing Center (NHR), where each node consists of two sockets, each with ten physical CPU cores. By pinning each process to the ten cores of a separate socket, only four individuals per generation need to be evaluated per process, which reduces the required time to run an optimization to less than 24 h in all considered test cases.

5.2 Poisson's equation

Poisson's equation is an elliptic partial differential equation defined by

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega \\ u &= g \quad \text{on } \partial\Omega. \end{aligned} \tag{15}$$

We consider two different instances of Eq. (15) with Dirichlet boundary conditions, summarized in Table 3. In both cases, we discretize the Laplace operator ∇^2 with finite differences on a uniform cartesian grid of size $h = 1/l_{max}$, which in two dimensions yields the five point stencil

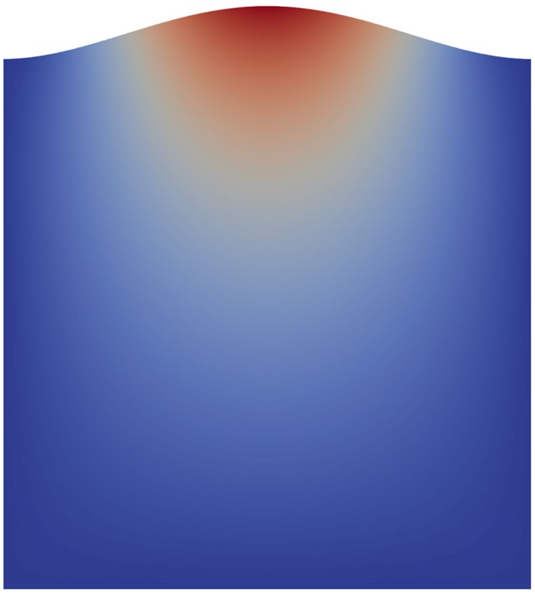
Fig. 2 Visualization of the considered linear elastic boundary value problem. A two-dimensional rectangular body undergoes an elastic deformation into y-direction

$$\left(\nabla^2 u\right)_{i,j} = \frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}),$$

and in three dimensions the seven point stencil

$$\left(\nabla^2 u\right)_{i,j,k} = \frac{1}{h^2}(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} - 6u_{i,j,k}),$$

whereby we choose $l_{max} = 11$ in the two-dimensional and $l_{max} = 7$ in the three-dimensional case. The resulting systems of linear equations then consist of 4 190 209 and 2 048 383 unknowns, respectively.

5.3 Linear elasticity

Linear elasticity is an essential branch of solid mechanics, characterized as a linear relationship between stress and strain, that has numerous applications in engineering and material science [18]. We consider a two-dimensional linear elastic boundary value problem, formulated in the form of the following system of PDEs, which models a two-dimensional rectangular body that undergoes an elastic deformation into y-direction, as it can be seen in Fig. 2:

$$\begin{aligned} (\alpha + \beta) \cdot \left(\frac{\partial^2}{\partial x^2} u + \frac{\partial^2}{\partial x \partial y} v \right) + \alpha \nabla^2 u = 0 \quad & \text{in } \Omega \\ (\alpha + \beta) \cdot \left(\frac{\partial^2}{\partial x \partial y} u + \frac{\partial^2}{\partial y^2} v \right) + \alpha \nabla^2 v = 0 \quad & \text{in } \Omega \\ u = 0 \quad \text{and} \quad v = g \quad & \text{on } \partial\Omega \end{aligned} \tag{16}$$

where $\Omega = (0,1)^2$, $\alpha = 195$, $\beta = 130$ and

$$g(x,y) = 0.4\,(1-x)\,xy\,\sin(\pi x).$$

We discretize Eq. (16) using finite differences on a cartesian grid with a step size of h, to obtain the system of linear equations $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f}$ with

$$\boldsymbol{A} = \begin{pmatrix} (\alpha+\beta)\frac{\partial^2}{\partial x^2} + \alpha\nabla^2 & (\alpha+\beta)\frac{\partial^2}{\partial x \partial y} \\ (\alpha+\beta)\frac{\partial^2}{\partial x \partial y} & (\alpha+\beta)\frac{\partial^2}{\partial y^2} + \alpha\nabla^2 \end{pmatrix},$$
$$\boldsymbol{u} = \begin{pmatrix} u \\ v \end{pmatrix}, \quad \boldsymbol{f} = \begin{pmatrix} f_u \\ f_v \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

whereby the differential operators ∇^2, $\frac{\partial^2}{\partial x^2}$, $\frac{\partial^2}{\partial y^2}$ and $\frac{\partial^2}{\partial x \partial y}$ are approximated by their discrete counterparts

$$\begin{aligned}
\left(\nabla^2 u\right)_{i,j} &= \frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) \\
\left(\frac{\partial^2}{\partial x^2} u\right)_{i,j} &= \frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} - 2u_{i,j}) \\
\left(\frac{\partial^2}{\partial y^2} u\right)_{i,j} &= \frac{1}{h^2}(u_{i,j-1} + u_{i,j+1} - 2u_{i,j}) \\
\left(\frac{\partial^2}{\partial x \partial y} u\right)_{i,j} &= \frac{1}{4h^2}(u_{i+1,j+1} + u_{i-1,j-1} - u_{i-1,j+1} - u_{i+1,j-1}).
\end{aligned}$$

Similar to the above case, we employ a uniform cartesian grid of size $h = 1/l_{max}$ with $l_{max} = 10$, such that the resulting system of linear equations contains 2 093 058 unknowns.

6 Results and discussion

To evaluate whether our optimization approach can consistently construct efficient multigrid solvers, we have performed ten independent experiments for each of the three considered cases. The Figs. 3, 4 and 5 show the mean and standard deviation of the current optima for both objectives during the optimization in all of the experiments performed. First, the question arises whether our algorithm can effectively minimize the values of both objective functions during the optimization. By investigating Figs. 3a, 4a and 5a it becomes apparent that, in general, our algorithm is able to drastically reduce the minimum convergence factor within the first 100 generations. The same is the case for the second objective, the execution time per iteration of a solver. However, as Figs. 3b, 4b and 5b shows, the majority of decrease is already achieved within the first 50 generations of the optimization. In general, we can observe that in all three cases, the optimization of the convergence factor requires more generations, and significant

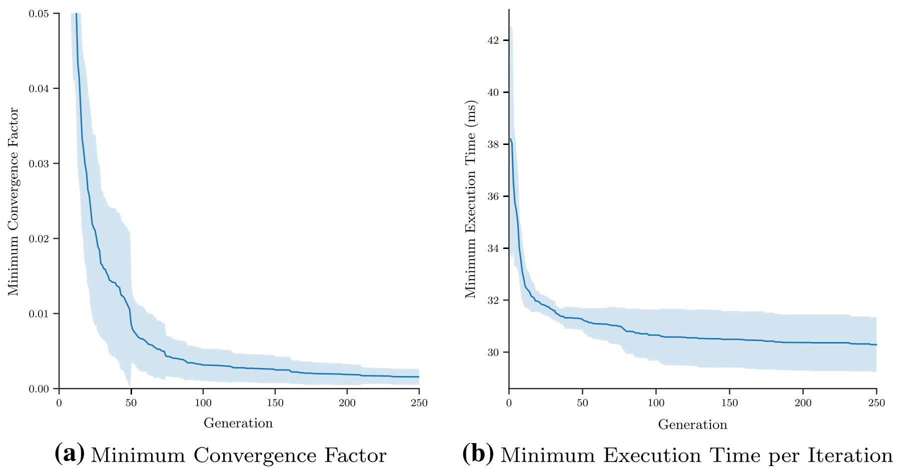

Fig. 3 2D Poisson–Mean and standard deviation of the minimum objective function values during the optimization

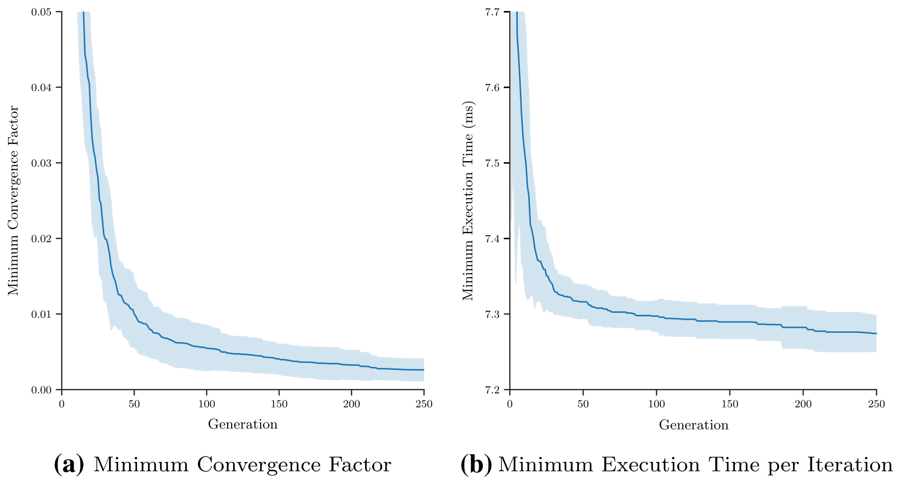

Fig. 4 3D Poisson–Mean and standard deviation of the minimum objective function values during the optimization

reductions still occur beyond 100 generations. Furthermore, by investigating the range of values achieved for the first objective, we can assess the difficulty of the underlying problem. While the execution time per iteration is solely determined by the computational complexity of the individual operations employed within a solver, for an easier problem, faster convergence, and therefore a smaller convergence factor can be attained. As known from multigrid theory [34], the two- and three-dimensional Poisson's equation represent relatively easy problems for the

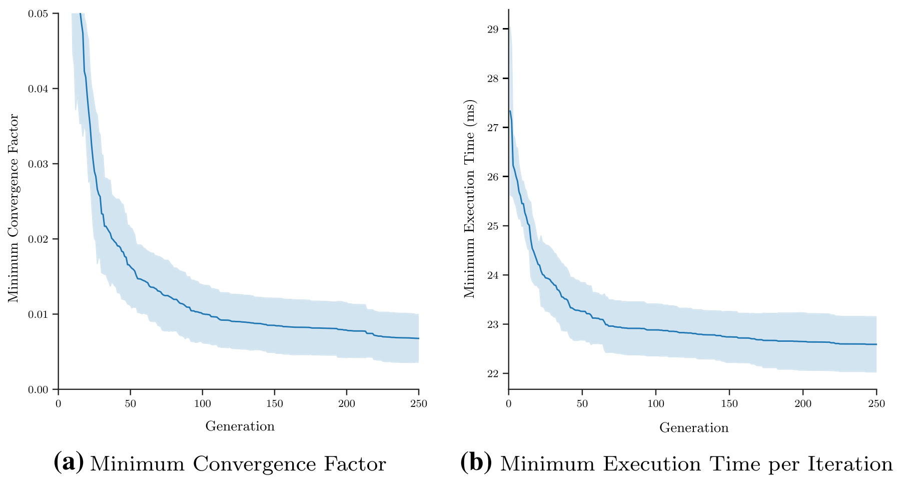

Fig. 5 2D Linear Elasticity–Mean and standard deviation of the minimum objective function values during the optimization

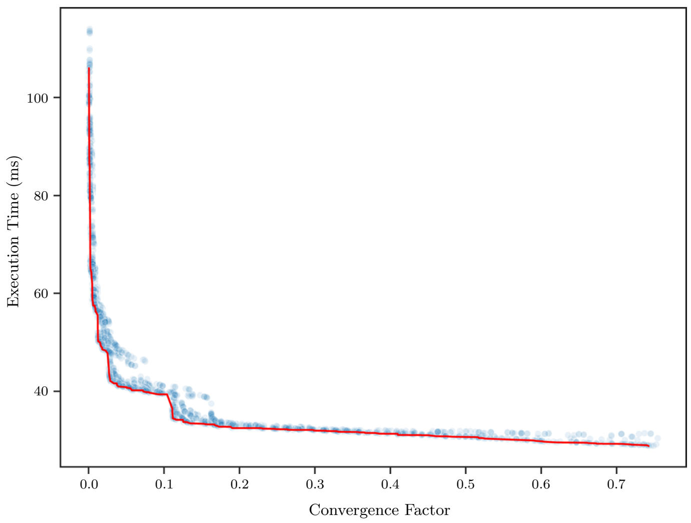

Fig. 6 2D Poisson–Pareto distribution at the end of all ten experiments. The red line denotes the combined Pareto front

construction of multigrid solvers, and hence the mean optimum convergence factor falls below a value of 0.005. For the linear elastic boundary value problem, both the mean and standard deviation is higher. However, on average, we can still construct multigrid methods that achieve a convergence factor of 0.01 or less, which represents an exceptionally fast convergence. Finally, we can conclude

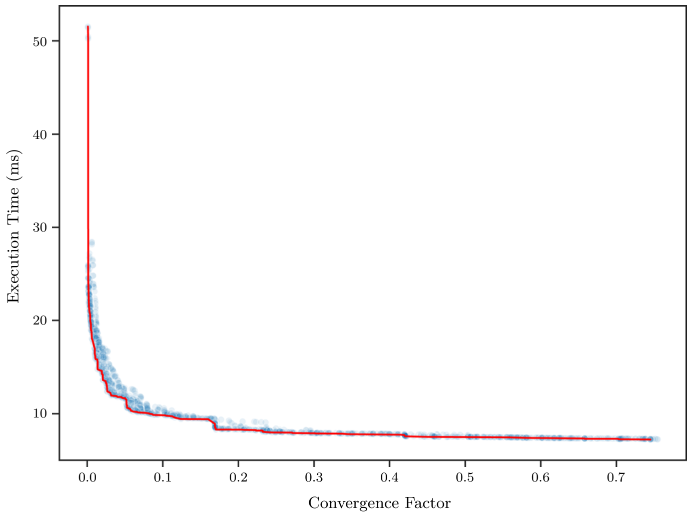

Fig. 7 3D Poisson–Pareto distribution at the end of all ten experiments. The red line denotes the combined Pareto front (Color figure online)

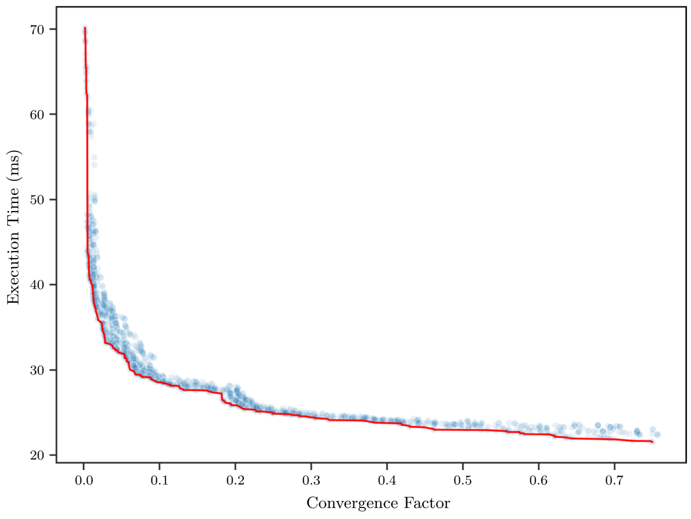

Fig. 8 2D Linear Elasticity–Pareto distribution at the end of all ten experiments. The red line denotes the combined Pareto front (Color figure online)

that our optimization approach is able to consistently find satisfactory minima for both objectives for all three problems considered.

6.1 Pareto distribution analysis

To further assess the outcome of our multi-objective optimization, the Figs. 6, 7 and 8 show the combined Pareto distributions of all ten experiments. Here, the red curve represents the resulting Pareto front, while the combined density of the data points indicates where the majority of the solutions is located. In all three cases, the objective function values of most individuals are close to the combined Pareto front, whereby the number of individuals is overall higher in the center of the front, i.e., the lower left part of the objective function space. In principle, the solutions located there represent a compromise between the two objectives and are, hence, the most promising solver candidates. In Fig. 6 the number of individuals that are distinctly located outside the Pareto front is slightly higher than in the other two cases, although, compared to the complete objective function space, the minimal distance from these outliers to the next point located on the combined Pareto front is still comparably small. In general, we can conclude that, under the given conditions, our optimization algorithm can consistently evolve a similar Pareto front in the majority of the performed experiments for all three considered problems. However, it must be noted that the employed population size is not sufficient to evolve a set of Pareto-optimal candidate solutions that are evenly distributed over the objective space, which can be attributed to the vast size of the search space as discussed in Sect. 3.3. While our approach's scalability, in principle, supports the evaluation of a significantly larger number of individuals than considered here through the use of distributed computing capabilities, the availability of computational resources is limited. Although, it can be expected that future architectural advances and performance improvements will enable the application of our approach to larger population sizes and problems with higher computational requirements.

6.2 Comparison with reference methods

Finally, since our goal is to automatically construct multigrid solvers that are competitive with renowned methods developed within decades of mathematical research, we evaluate their efficiency on the three test problems on two different evaluation platforms and compare them with several hand-crafted multigrid cycles. We consider two multi-core CPU architectures for evaluation: Intel Xeon E5-2630v4 (Broadwell) and Intel Xeon 2660v2 (Ivy Bridge). In both cases, we execute each solver on a single compute node, consisting of two sockets, with a total number of 20 physical cores. So far, we have only considered a single problem size for each test problem, i.e. $l_{max} = 11$ for the two-dimensional, $l_{max} = 7$ for the three-dimensional Poisson equation and $l_{max} = 10$ for the linear elastic boundary value problem. However, an essential property of multigrid methods is to achieve the same degree of efficiency on larger problem instances. For this purpose, we also evaluate each solver on a larger instance of the respective test

Table 4 2D Poisson—Measured number of iterations and solving times of the reference methods on 20 cores and two sockets

l_{max}	Iterations		Broadwell (ms)		Ivy Bridge (ms)	
	11	12	11	12	11	12
V(1, 0)	21	21	969	2810	879	2652
V(1, 1)	9	9	461	1359	411	1287
V(2, 1)	7	7	377	1137	334	1087
V(2, 2)	6	6	344	1056	302	1007
V(3, 2)	6	6	378	1160	324	1112
V(3, 3)	6	6	397	1255	344	1201
V(4, 3)	6	6	425	1350	366	1306
V(4, 4)	6	6	448	1449	383	1409

Table 5 3D Poisson - Measured number of iterations and solving times of the reference methods on 20 cores and two sockets

l_{max}	Iterations		Broadwell (ms)		Ivy Bridge (ms)	
	7	8	7	8	7	8
V(1, 0)	29	30	121.3	1221	134.6	1470
V(1, 1)	13	13	70.8	682	79.9	838
V(2, 1)	9	9	59.0	582	66.2	708
V(2, 2)	7	7	54.6	531	65.4	654
V(3, 2)	7	7	61.9	610	74.6	757
V(3, 3)	7	7	72.6	690	86.6	857
V(4, 3)	7	6	77.9	656	87.3	825
V(4, 4)	6	6	73.2	725	82.5	906

Table 6 2D Linear Elasticity - Measured number of iterations and solving times of the reference methods on 20 cores and two sockets

l_{max}	Iterations		Broadwell (ms)		Ivy Bridge (ms)	
	10	11	10	11	10	11
V(1, 0)	32	31	872	4306	828	4128
V(1, 1)	15	15	439	2118	418	2075
V(2, 1)	10	10	318	1529	312	1529
V(2, 2)	9	9	314	1449	316	1476
V(3, 2)	8	8	297	1368	304	1388
V(3, 3)	7	7	283	1247	288	1288
V(4, 3)	7	7	293	1320	313	1397
V(4, 4)	7	7	311	1378	334	1471

problem. As a baseline for multigrid solver efficiency, we consider several different reference methods. Besides full multigrid, which we do not consider in this work, V-cycles with overrelaxed red-black Gauss-Seidel smoothing represent

Table 7 2D Poisson - Measured number of iterations and solving times of the evolved multigrid methods on 20 cores and two sockets

l_{max}	Iterations		Broadwell (ms)		Ivy Bridge (ms)	
	11	12	11	12	11	12
ES-1	5	5	338	1064	304	1055
ES-2	6	6	371	1163	330	1133
ES-3	5	5	311	988	279	976
ES-4	6	6	380	1188	338	1153
ES-5	5	5	312	978	279	963
ES-6	5	5	349	1123	309	1106
ES-7	6	6	354	1096	320	1068
ES-8	6	6	347	1081	310	1056
ES-9	6	6	353	1079	313	1045
ES-10	5	5	310	960	275	934

the most efficient multigrid methods for solving the discretized Poisson's equation [34]. As we have already investigated in [32], the same is also true for the considered linear elastic boundary value problem. In all three cases, we choose an optimal relaxation factor for the smoother from the same interval considered within the optimization, which leads to $\omega = 1.15$ for the two-dimensional Poisson equation and $\omega = 1.25$ both for the three-dimensional Poisson equation and the linear elastic boundary value problem. The Tables 4, 5 and 6 contain the required number of iterations and solving times to achieve the desired defect reduction for the three test problems with the two considered problem sizes. For instance, the abbreviation V(2, 1) denotes a V-cycle with two pre- and one post-smoothing step with red-black Gauss-Seidel. Note that in all three cases, the number of iterations stays almost constant for both problem sizes. In general, we can identify the V(2, 2)-cycle as the most efficient solver both for the two- and three-dimensional Poisson equation and the V(3, 3)-cycle as the most efficient solver for the linear elastic boundary value problem, which is the case for both considered CPU architectures.

As the last step, we evaluate the solvers constructed with our optimization approach under the same conditions. Since the number of individuals contained in the Pareto front varies and can potentially be too large for a direct evaluation of all contained individuals, we heuristically identify the 50 most promising solvers. For this purpose, we sort the Pareto front according to the metric

$$T_\varepsilon = \frac{\log(\varepsilon)}{\log(\tilde{\rho})} \cdot t, \tag{17}$$

where $\varepsilon = 10^{-12}$ is the desired defect reduction factor and $\tilde{\rho}$ and t the objective function values obtained within the optimization. The resulting list of solvers is then evaluated on the 20 cores of a compute node with Broadwell architecture to identify the one with the lowest solving time, which is then considered for all subsequent

Table 8 3D Poisson - Measured number of iterations and solving times of the evolved multigrid methods on 20 cores and two sockets

l_{max}	Iterations		Broadwell (ms)		Ivy Bridge (ms)	
	7	8	7	8	7	8
ES-1	10	11	55.3	577	70.0	704
ES-2	8	9	57.2	578	64.3	716
ES-3	8	9	59.0	671	65.3	824
ES-4	8	9	54.6	576	62.7	710
ES-5	8	10	54.6	641	60.9	789
ES-6	9	10	59.4	716	67.1	891
ES-7	6	8	56.2	702	70.9	880
ES-8	5	5	56.7	589	74.0	724
ES-9	10	10	61.0	568	66.3	681
ES-10	10	11	55.4	581	61.3	705

Table 9 2D Linear Elasticity - Measured number of iterations and solving times of the evolved multigrid methods on 20 cores and two sockets

l_{max}	Iterations		Broadwell (ms)		Ivy Bridge (ms)	
	10	11	10	11	10	11
ES-1	6	6	234	1117	235	1137
ES-2	6	6	216	1033	211	1035
ES-3	7	7	258	1225	259	1231
ES-4	6	6	226	1077	219	1093
ES-5	6	6	235	1121	229	1139
ES-6	6	6	220	1083	213	1093
ES-7	7	7	238	1191	236	1186
ES-8	6	6	217	1037	223	1039
ES-9	6	6	224	1039	222	1058
ES-10	7	7	243	1188	238	1188

evaluations on both CPU architectures. While we could also identify the most promising solver on the Ivy Bridge CPU, we focus on Broadwell, as it represents a more recent design. The Tables 7, 8 and 9 contain the resulting measurements for the evolved solvers ES-[1-10], which have been chosen from the Pareto front at the end of each optimization run with the above heuristic. In general, all constructed solvers represent functioning multigrid methods for both considered problem sizes in all three investigated cases. However, the achieved solving time differs between the individual experiments, which is especially the case for the three-dimensional Poisson problems, where our approach cannot consistently obtain solvers with the same degree of efficiency as in the other two cases. If we compare the average solving times achieved in all three cases, it becomes apparent that for the three-dimensional Poisson equation with $l_{max} = 7$ the difference between the individual solvers is in the order of magnitude of a few milliseconds. Therefore, the use of this problem size within the optimization hampers the identification of Pareto-optimal solvers since it is impossible to eliminate the influence of CPU performance variations within our code generation-based evaluation. For three dimensional problems the number of unknowns increases cubically with the problem size $n = 1/h - 1 = 2^l - 1$, which means that for $l_{max} = 8$ we have to solve a system with 16 581 375 unknowns for the evaluation of each solver within the optimization. Since the cost of performing code generation also increases for three-dimensional problems, so far, we could not consider larger instances within our approach. However, in the majority of experiments, the constructed solvers still represent efficient methods for both considered problem sizes of the three-dimensional Poisson equation, whereby the most efficient solver for $l_{max} = 8$, ES-9, achieves a faster solving time than the second-best reference method, the V(2, 1)-cycle, on both CPU architectures. In contrast, for both two-dimensional PDEs, our optimization approach manages to construct solvers that are more efficient than the best reference method for both problem sizes and CPU architectures. For the two-dimensional Poisson equation, in three of the ten experiments, ES-3, ES-5, and ES-10, we were able to construct solvers that achieve consistently faster solving times than the V(2, 2)-cycle, whereby the achieved speedup ranges from a few percent up to almost ten. The two-dimensional Poisson equation has been thoroughly studied and represents a standard test case for the application of multigrid. Therefore, the fully automatic construction of solvers that outperform multigrid cycles developed in decades of mathematical research for different problem sizes already represents a significant achievement. Even beyond that, we can construct consistently faster multigrid solvers for the considered two-dimensional linear elastic boundary value problem than the most efficient reference method, the V(3, 3)-cycle, in each of the ten experiments. For instance, the constructed solver, ES-2, solves the given problem 17 % faster for $l_{max} = 11$ and 23 % faster for $l_{max} = 10$ than the mentioned V-cycle on the Broadwell evaluation platform, while even slightly higher speedups can be achieved on Ivy Bridge.

7 Conclusion

In this work, we have laid the foundations for the automatic construction of efficient geometric multigrid solvers based on a tailored context-free grammar and the use of evolutionary search methods. It, therefore, opens up the possibility of applying the field of grammar-guided genetic programming (GGGP) to the optimization of multigrid methods. Furthermore, while in [32] we could already demonstrate that this approach is capable of constructing functioning multigrid solvers for a linear elastic boundary value problem, the outcome was still limited by the accuracy of the models used for the prediction of a solver's efficiency. In this work, we have been able to overcome this limitation through a distributed code generation-based solver evaluation and, hence, could demonstrate the construction of multigrid solvers that are able to outperform efficient reference methods both in the previously considered linear elastic boundary value problem, as well as a two-dimensional Poisson problem. While we could not achieve the same degree of efficiency for a three-dimensional Poisson problem, the constructed solvers still represent functioning and efficient multigrid methods. In addition, for the first time, we could also demonstrate that the solvers constructed through GGGP can achieve similar performance for a larger instance of the investigated problems. Achieving generalization, i.e., designing an algorithm that is not only able to solve a single problem instance but can deal with an entire class of problems, is a fundamental goal of artificial intelligence-based algorithm design. For many PDE-based applications, for instance, saddle point problems [2], the construction of a general and efficient multigrid solver has not yet been demonstrated, which leaves room for a wide range of extensions of the approach presented in this work. Furthermore, in this and our previous work, we have only considered classical geometric multigrid methods based on the original formulation by Brandt [4]. However, the mathematical properties of particular problems prohibit the use of such a method [11] but require alternative approaches to construct an efficient multigrid-based numerical solution method, for example, using multigrid as a preconditioner to accelerate the convergence of Krylov subspace methods [10, 34]. Also, many PDEs, such as the Navier-Stokes equation, are substantially nonlinear and, hence, require an adaption of the classical multigrid formulation to deal with the occurrence of these nonlinearities, for instance, in the form of Newton-multigrid methods or the full-approximation scheme (FAS) [5, 34]. Finally, a different aspect, which has been already mentioned in [32], is combining our approach with the complementary branch of machine learning-based methods by incorporating optimized prolongation [15, 22] and smoothing operators [20] into our formal grammar.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. T. Back, D.B. Fogel, Z. Michalewicz, *Handbook of Evolutionary Computation*, 1st edn. (IOP Publishing Ltd., GBR, Bristol, 1997)
2. M. Benzi, G.H. Golub, J. Liesen, Numerical solution of saddle point problems. Acta Numer. **14**, 1–137 (2005)
3. H.G. Beyer, H.P. Schwefel, Evolution strategies - a comprehensive introduction. Nat. Comput. **1**(1), 3–52 (2002)
4. A. Brandt, Multi-level adaptive solutions to boundary-value problems. Math. Comput. **31**(138), 333–390 (1977)

5. W.L. Briggs, V. E. Henson, S.F. McCormick, *A multigrid tutorial*, 2nd esdn, (Society for Industrial and Applied Mathematics, Philadelphia, 2000)
6. J. Brown, Y. He, S. MacLachlan, M. Menickelly, S.M. Wild, Tuning multigrid methods with robust optimization and local fourier analysis. SIAM J. Sci. Comput. **43**(1), A109–A138 (2021). https://doi.org/10.1137/19M1308669
7. C.A.C. Coello, G.B. Lamont, D.A. Van Veldhuizen et al., *Evolutionary Algorithms for Solving Multi-Objective Problems*, vol. 5 (Springer-Verlag, New York, 2007)
8. K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: Nsga-ii. In: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature, PPSN VI, (Springer-Verlag, Berlin, Heidelberg, 2000), p. 849–858. https://doi.org/10.1007/3-540-45356-3_83
9. J. Demmel, I. Dumitriu, O. Holtz, Fast linear algebra is stable. Numer. Math. **108**(1), 59–91 (2007). https://doi.org/10.1007/s00211-007-0114-x
10. Y.A. Erlangga, C.W. Oosterlee, C. Vuik, A novel multigrid based preconditioner for heterogeneous helmholtz problems. SIAM J. Sci. Comput. **27**(4), 1471–1492 (2006)
11. O.G. Ernst, M.J. Gander, Why it is Difficult to Solve Helmholtz Problems with Classical Iterative Methods. In: I.G. Graham, T. Hou, O. Lakkis, R. Scheichl (eds.) Numerical Analysis of Multiscale Problems (Springer, Berlin, Heidelberg, 2012), p. 325–363. https://doi.org/10.1007/978-3-642-22061-6_10
12. R.P. Fedorenko, A relaxation method for solving elliptic difference equations. Zhurnal Vychislitel'noi Mat. Mat. Fiziki **1**(5), 922–927 (1961)
13. F.A. Fortin, F.M. De Rainville, M.A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy. J. Mach. Learn. Res. **13**, 2171–2175 (2012)
14. F.A. Fortin, S. Grenier, M. Parizeau, Generalizing the improved run-time complexity algorithm for non-dominated sorting. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13, (Association for Computing Machinery, New York, NY, USA, 2013), p. 615–622. https://doi.org/10.1145/2463372.2463454
15. D. Greenfeld, M. Galun, R. Basri, I. Yavneh, R. Kimmel, Learning to optimize multigrid pde solvers. In: International Conference on Machine Learning, 2415–2423 (2019)
16. W. Hackbusch, *Multi-Grid Methods and Applications* (Springer-Verlag, New York, 1985)
17. N. Hansen, A. Ostermeier, Completely derandomized self-adaptation in evolution strategies. Evol. Comput. **9**(2), 159–195 (2001). https://doi.org/10.1162/106365601750190398
18. G. Holzapfel, *Nonlinear Solid Mechanics: A Continuum Approach for Engineering* (John Wiley and Sons, Hoboken, 2001)
19. J.T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, S. Ermon, Learning neural pde solvers with convergence guarantees. arXiv preprint (2019)
20. R. Huang, R. Li, Y. Xi, Learning optimal multigrid smoothers via neural networks. arXiv preprint arXiv:2102.12071 (2021)
21. K. Kahl, N. Kintscher, Automated local fourier analysis (alfa). BIT Numer. Math. **60**(3), 651–686 (2020)
22. A. Katrutsa, T. Daulbaev, I. Oseledets, Black-box learning of multigrid parameters. J. Comput. Appl. Math. **368**, 112524 (2020)
23. J.R. Koza, Genetic programming as a means for programming computers by natural selection. Stat. Comput. **4**(2), 87–112 (1994). https://doi.org/10.1007/BF00175355
24. J.R. Koza, Human-competitive results produced by genetic programming. Genet. Program. Evolvable Mach. **11**(3–4), 251–284 (2010). https://doi.org/10.1007/s10710-010-9112-3
25. C. Lengauer, S. Apel, M. Bolten, S. Chiba, U. Rüde, J. Teich, A. Größlinger, F. Hannig, H. Köstler, L. Claus, et al., Exastencils: advanced multigrid solver generation. In: Software for Exascale Computing - SPPEXA 2016–2019, (Springer, Cham, 2012), p. 405–452.
26. R.I. Mckay, N.X. Hoai, P.A. Whigham, Y. Shan, M. O'neill, Grammar-based genetic programming: a survey. Genet. Program. Evolvable Mach. **11**(3–4), 365–396 (2010)
27. C.W. Oosterlee, R. Wienands, A genetic search for optimal multigrid components within a fourier analysis setting. SIAM J. Sci. Comput. **24**(3), 924–944 (2003). https://doi.org/10.1137/S1064827501397950
28. H. Rittich, Extending and automating fourier analysis for multigrid methods. Ph.D. thesis, University of Wuppertal (2017)
29. Y. Saad, Iterative Methods for Sparse Linear Systems, 2 edn, (Society for Industrial and Applied Mathematics, Philadelphia, 2003)

30. C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, J. Teich, Exaslang: a domain-specific language for highly scalable multigrid solvers. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, 42–51. IEEE (2014)
31. C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, H. Köstler, U. Rüde, C. Lengauer, Systems of partial differential equations in exaslang, in *Software for Exascale Computing-SPPEXA 2013-2015*. (Springer, New York, 2016), pp. 47–67
32. j. Schmitt, S. Kuckuk, H Köstler, Constructing efficient multigrid solvers with genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO '20, 1012–1020. (Association for Computing Machinery, New York, NY, USA, 2020). https://doi.org/10.1145/3377930.3389811
33. A. Thekale, T. Gradl, K. Klamroth, U. Rüde, Optimizing the number of multigrid cycles in the full multigrid algorithm. Numer. Linear Algebra Appl. **17**(2–3), 199–210 (2010)
34. U. Trottenberg, C.W. Oosterlee, A. Schuller, *Multigrid* (Elsevier, Amsterdam, 2000)
35. R. Wienands, W. Joppich, *Practical Fourier Analysis for Multigrid Methods* (CRC Press, Florida, 2004)
36. S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785